9. Funzioni
(parte seconda)

Andrea Marongiu
(andrea.marongiu@unimore.it)
Paolo Valente

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Domanda

- Se una funzione
  - lavora su un certo valore in ingresso
  - e, quando si progetta la funzione, si può scegliere tra
    - Far leggere tale valore alla funzione da *stdin*?
    - Far ricevere tale valore in ingresso dalla funzione attraverso un parametro formale
- Quale delle due soluzioni è migliore?

Soluzione migliore ingresso 1/2

- La soluzione migliore è la seconda
  - La funzione può essere utilizzata ovunque all'interno del programma passandole il valore che si preferisce
  - Non è necessario dover leggere obbligatoriamente qualcosa da *stdin*
  - Eventuali letture da *stdin* si possono effettuare semplicemente nel main o in generale in altre funzioni il cui scopo è proprio quello di leggere qualcosa da stdin

Soluzione migliore ingresso 2/2

- Pensate ad esempio all'uso della funzione sqrt nel programma che verifica se un numero è primo
- Se la funzione non avesse avuto un parametro formale tramite il quale passarle il valore su cui lavorare
  - Ma lo avesse letto da *stdin*
- Come saremmo riusciti a scrivere il programma in maniera tale che chiedesse, correttamente, una sola volta il numero su cui lavorare?
  - Ossia il numero del quale stabilire se fosse primo oppure no

Risposta

- Non ci sarebbe stato **alcun modo**

Domanda

- Se una funzione
  - fornisce un certo valore in uscita
  - e, quando si progetta la funzione, si può scegliere tra
    - Far scrivere tale valore su *stdout*
    - Far restituire tale valore alla funzione attraverso l'istruzione *return*
- Quale delle due soluzioni è migliore?

Soluzione migliore uscita 1/2

- La soluzione migliore è la seconda
  - La funzione può essere utilizzata ovunque all'interno del programma, leggendo ed eventualmente memorizzando in una variabile il valore da essa ritornato
  - Non è necessario dover necessariamente stampare qualcosa su *stdout*
  - Eventuali scritture su *stdout* si possono effettuare semplicemente nel *main* o in generale in altre funzioni il cui scopo è proprio quello di scrivere qualcosa su *stdout*

Soluzione migliore uscita 2/2

- Pensate di nuovo all'uso della funzione *sqrt* nel programma che verifica se un numero è primo
- Se la funzione non avesse ritornato la radice quadrata del numero passato in ingresso
  - Ma avesse stampato il risultato su *stdout*
- Saremmo riusciti a scrivere il programma?

Risposta

- No

Il main è una funzione

- La prima istruzione della funzione **main** è la prima istruzione dell'intero programma
- Le variabili definite nella funzione **main** hanno valori casuali
- Quando la funzione **main** termina, tutto il programma termina
- In un programma corretto, la funzione **main** ha tipo di ritorno **int**
- Il valore intero ritornato dalla funzione **main** coincide col valore restituito dal processo quando termina

Chiamate incrociate 1/2

```
void fun1()
{
     ...
     fun2();
     ...
}

void fun2()
{
     ...
     fun1();
     ...
}
```

- *Ancora non è stata definita!*
- *Invertire l'ordine di definizione delle funzioni risolverebbe il problema?*

Chiamate incrociate 2/2

```
void fun1()
{
     ...
     fun2();
     ...
}

void fun2()
{
     ...
     fun1();
     ...
}
```

- *Purtroppo no ...*

Dichiarazione 1/2

- Come abbiamo già detto a suo tempo, una definizione è una caso particolare di dichiarazione
- In particolare:
  - Una definizione di variabile o costante con nome è una dichiarazione che causa l'allocazione di spazio in memoria quando viene incontrata
  - Una definizione di funzione è un caso particolare di dichiarazione in cui si definisce il corpo della funzione

Dichiarazione 2/2

- In generale, una dichiarazione è una istruzione in cui si introduce un nuovo identificatore e se ne dichiara il tipo
- In C/C++ ogni identificatore si può utilizzare solo dopo essere stato dichiarato
- Quindi le definizioni sono delle dichiarazioni in cui non solo si introduce un nuovo identificatore ed il tipo associato, ma
  - nel caso delle variabili e costanti con nome si alloca anche memoria
  - nel caso delle funzioni si definisce anche il corpo della funzione
- Vediamo quindi la dichiarazione senza definizione di una funzione

Dichiarazione funzione

- Una **dichiarazione** (senza definizione) o **prototipo** di una funzione è costituita dalla sola intestazione di una funzione seguita da un punto e virgola

<dichiarazione-funzione> ::= *<intestazione-funzione>* **;**

<intestazione-funzione> ::=
<nomeTipo> *<nomeFunzione>* **(** *<lista-parametri>* **)**

<lista-parametri> ::=
<nessun carattere> | **`void`** |
<dich-parametro> { **,** *<dich-parametro>* }

<dich-parametro> ::=
[**`const`**] *<nomeTipo>* [*<identificatore>*]

Opzionale !

Esempi di prototipi

```
int  fattoriale (int);

main()
{
      … // invocazione funzione fattoriale
}

int  fattoriale   (int n)
{
      int fatt=1;
      for (int i=1; i<=n; i++)
            fatt = fatt*i;
      return(fatt);
 }
```

```
int massimo (int, int, int) ;  /* calcola il max di 3 int */
```

Soluzione chiamate incrociate

```
void fun2() ; // dichiarazione di fun2

void fun1()
{
     ...
     fun2();
     ...
}

void fun2()
{
     ...
     fun1();
     ...
}
```

Prototipi e definizioni

- Il prototipo:
  - è un puro “avviso ai naviganti”
  - **non causa la produzione di alcun byte di codice eseguibile**
  - può essere ripetuto più volte nel programma
  - (basta che non ci siano due dichiarazioni in contraddizione)
  - può comparire anche dentro un'altra funzione (non usiamolo in questo modo)
-
- La definizione, invece:
  - contiene il codice della funzione
  - non può essere duplicata!!
  - (altrimenti ci sarebbero due codici per la stessa funzione)
  - non può essere inserita in un'altra funzione
  - il nome dei parametri formali, **non necessario in un prototipo**, è importante in una definizione
-
- QUINDI: il **prototipo** di una funzione può **comparire più volte**, ma la funzione deve essere **definita una sola volta**

Domanda

- Quali elementi di una chiamata di funzione deve controllare il compilatore per essere sicuro che la funzione sia invocata in modo sintatticamente corretto?

Risposta

- Numero di parametri attuali
  - Deve essere uguale al numero di parametri formali
- Tipo di ciascun parametro attuale
  - Il tipo di ciascun parametro attuale deve essere compatibile col tipo del parametro formale nella posizione corrispondente
- Tipo del valore atteso nel punto del programma in cui si utilizza il valore di ritorno della funzione
  - Tale valore di ritorno non si può utilizzare affatto se la funzione ha tipo di ritorno *void*

Domanda

- Cosa è sufficiente conoscere, da parte del compilatore, per accertarsi che l'invocazione di una funzione sia sintatticamente corretta in merito agli aspetti evidenziati nella precedente slide?

Risposta

- Tutte e sole le informazioni contenute nell'intestazione, ossia nella dichiarazione, della funzione!
  - Numero dei parametri formali
  - Tipo di ciascun parametro formale
  - Tipo di ritorno della funzione

Controllo sintattico invocazione

- La precedente risposta è il motivo fondamentale per cui l'unico vincolo posto dal compilatore per poter inserire correttamente l'invocazione di una funzione in un dato punto del programma è che la dichiarazione della funzione preceda, nel testo del programma, il punto in cui la funzione è invocata
- Tale vincolo è imposto per aumentare la capacità del compilatore di **trovare subito** errori *sintattici* commessi dal programmatore
  - Spesso tali errori sintattici scaturiscono da errori concettuali, che vengono così rilevati immediatamente

Definizione corpo

- Al compilatore interessa in prima battuta di controllare solo la correttezza sintattica delle invocazioni, e per questo bastano solo le intestazioni delle funzioni come abbiamo visto
- In quanto al corpo, ossia al codice vero e proprio di una funzione, al compilatore basta solo trovare prima o poi, nel testo del programma, la definizione di tale corpo (che verrà tradotto in linguaggio macchina)
  - Quando lo trova, traduce ciascuna invocazione della funzione in un *salto* all'esecuzione di tale corpo (ed altre operazioni accessorie)
  - Se non lo trova, allora segnala un errore
- Si può quindi definire il corpo di una funzione dove si vuole nel testo del programma

Esempio di programma errato

```
main()
{
  int a, b;
  cin>>a>>b ;
  cout<<"Il massimo tra "<<a<<" e "<<b<<" e' "
      <<massimo(a,b)<<endl ;
}

int massimo(int a, int b)
{
  if (a > b)
    return a ;
  return b ;
}
```

Versione corretta 1

```
int massimo(int a, int b)
{
  if (a > b)
    return a ;
  return b ;
}

main()
{
  int a, b;
  cin>>a>>b ;
  cout<<"Il massimo tra "<<a<<" e "<<b<<" e' "
      <<massimo(a,b)<<endl ;
}
```

Versione corretta 2

```
int massimo(int, int) ;

main()
{
  int a, b;
  cin>>a>>b ;
  cout<<"Il massimo tra "<<a<<" e "<<b<<" e' "
      <<massimo(a,b)<<endl ;
}

int massimo(int a, int b)
{
  if (a > b)
    return a ;
  return b ;
}
```

Tipo dei parametri.
Scrivere, ad esempio,
`int massimo(int a, int c);`
sarebbe stato equivalente

Parametri attuali
(*espressioni*)

Parametri formali
(definizioni di *variabili*)

Esercizio

- Scrivere una funzione che verifichi se un numero naturale passato in ingresso come parametro attuale è primo
  - Il numero **non viene letto da *stdin* da parte della funzione!**
- La funzione deve restituire falso se il numero non è primo, vero se il numero è primo
  - Attenzione al tipo di ritorno ...
- Utilizzando tale funzione, riscrivere il programma che controlla se due numeri primi sono gemelli

Soluzione funzione

```
bool isPrime(int n)
 {
   if (n>=1 && n<=3) return true; // 1,2,3: sì

   if (n%2==0) return false;     // no, perché pari

   for(int i=3, max_div = static_cast<int>(sqrt(n)) ;
       i<=max_div; i += 2)
          if (n%i==0)
              return false;   // no, perché è stato
                              // trovato un divisore

  // non è stato trovato alcun divisore
  return true;
 }
```

Resto del programma

```
main()
 {
   int n1, n2 ; cin>>n1>>n2 ;
   if (is_prime(n1) && is_prime(n2))
      if (n1 == n2 - 2 || n2 == n1 - 2)
         cout<<”n1 ed n2 sono due primi gemelli”<<endl ;
}
```

- Utilizzando la funzione abbiamo scritto in modo leggibile e senza replicazione del codice il nostro programma di verifica se due numeri sono primi gemelli

Passaggio dei parametri

- Per *passaggio dei parametri* si intende l'inizializzazione dei parametri formali di una funzione mediante i parametri attuali, che avviene al momento della chiamata della funzione
- L'unico meccanismo adottato in C, è il **PASSAGGIO PER VALORE**
- Come vedremo in lezioni successive, in C++ disponiamo anche del **passaggio per riferimento**

Passaggio per valore

- Le locazioni di memoria corrispondenti ai parametri formali:
  - Sono allocate al momento della chiamata della funzione
  - Sono inizializzate con i **valori** dei corrispondenti parametri attuali trasmessi dalla funzione chiamante
  - Vivono per tutto il tempo in cui la funzione è in esecuzione
  - Sono deallocate quando la funzione termina
  -
- QUINDI
  - La funzione chiamata effettua una **copia** dei valori dei parametri attuali passati dalla funzione chiamante
  - Tali copie sono sue copie private
  - Ogni modifica ai parametri formali è **strettamente locale alla funzione**
  - **I parametri attuali della funzione chiamante non saranno mai modificati!**

Esempio 1/2

```
int distanza_al_quadrato(int px1, int py1, int px2, int py2)
{
  px1 = pow (px1 - px2, 2); // pow(x, y) = x^y
  py2 = pow (py1 - py2, 2);
  return px1 + py2 ;
}

main()
{
  int a = 9, b = 9, c = 7, d = 12;

  cout<<a<<b<<c<<d<<endl;

  int dist =
       distanza_al_quadrato(a, b, c, d);

  cout<<a<<b<<c<<d<<dist<<endl;
}
```

Cosa viene stampato prima e dopo dell'invocazione di `distanza_al_quadrato`?

Esempio 2/2

```
int distanza_al_quadrato(int px1, int py1, int px2, int py2)
{
  px1 = pow (px1 - px2, 2);
  py2 = pow (py1 - py2, 2);
  return px1 + py2 ;
}

main()
{
  int a = 9, b = 9, c = 7, d = 12;

  cout<<a<<b<<c<<d<<endl;

  int dist =
  distanza_al_quadrato(a, b, c, d);

  cout<<a<<b<<c<<d<<dist<<endl;
}
```

Il collegamento tra parametri formali e parametri attuali si ha solo al momento della chiamata.
Sebbene px1 e py2 vengano modificati all'interno della funzione, i valori dei corrispondenti parametri attuali (a, d) rimangono inalterati.
Quindi gli stessi valori di a e d sono stampati prima e dopo

Esercizio

- Provare a scrivere una funzione che prenda in ingresso (come parametro formale) un numero naturale *n* e ritorni la somma dei numeri da 1 ad *n*
  - Definendo **una sola variabile locale** nella funzione
  - Senza utilizzare la chiusura della sommatoria
  - Quella che conterrà il risultato
- Per riuscirci bisogna utilizzare una tecnica sconsigliata
  - Facciamo questo esercizio solo per capire bene di cosa si tratti

Soluzione

```
int somma (int n)
{
  int somma = 0;
  // utilizzo decremento del parametro formale
  for (; n > 0; n--)
  somma += n;
  return somma;
}
```

Cosa viene stampato?

```
main()  {
  int risultato, n = 4 ;
  risultato = somma(n);
  cout<<"somma("<<n<<") = "<<risultato<<endl ;
}
```

Soluzione

```
int somma (int n)
{
  int somma = 0;
  // utilizzo decremento del parametro formale
  for (; n > 0; n--)
  somma += n;
  return somma;
}


main()  {
  int risultato, n = 4 ;
  risultato = somma(n);
  cout<<"somma("<<n<<") = "<<risultato<<endl ;
}
```

Anche se il parametro formale **n** viene modificato, la variabile **n** definita nel main *non viene alterata!* E' il suo valore (4) che viene passato alla funzione.

Stampa:
somma(4) = 10

Nota 1/2

- Abbiamo visto la modifica di un parametro formale variabile all'intero di una funzione solo per capire:
- 1) che la cosa si può fare, e 2) che tale parametro è perfettamente equivalente ad una variabile locale
- Tuttavia, è fondamentale avere presente che
  - In generale è una **cattiva abitudine** modificare i parametri formali per utilizzarli come variabili ausiliarie
    - Poca leggibilità: chi legge non capisce più se si tratta di parametri di ingresso (solo da leggere) o altro
    - Crea effetti collaterali nel caso di parametri passati per riferimento (che vedremo nelle prossime lezioni)

Nota 2/2

- L'**unico caso** in cui è necessario ed appropriato modificare i parametri formali è quando tali parametri sono intesi come **parametri di uscita**, ossia parametri in cui devono essere memorizzati valori che saranno poi utilizzati da chi ha invocato la funzione
- Questo **non può però accadere nel caso di passaggio per valore**, perché i parametri formali sono oggetti locali alla funzione, e saranno quindi eliminati alla terminazione della funzione stessa
- Vedremo più avanti come implementare i parametri di uscita mediante il passaggio per riferimento

Commenti passaggio per valore

- E’ sicuro: le variabili del chiamante e del chiamato sono ***completamente disaccoppiate***
- *Consente di ragionare per componenti isolati*: la struttura interna dei singoli componenti è irrilevante
- (la funzione può persino modificare i parametri ricevuti senza che ciò abbia alcun impatto sul chiamante)

- **LIMITI**
  - Impedisce *a priori* di scrivere funzioni che abbiano come scopo proprio quello di modificare variabili utilizzate poi nella funzione da cui sono invocate
  - Come vedremo il passaggio per valore può essere **costoso** per dati di **grosse dimensioni**

Domanda

- Se un parametro formale è dichiarato di tipo **`const`**, lo si può poi modificare all'interno della funzione?

Esempio:

```
int fun(const int j)
{
   j++ ;
}
```

Risposta

- Ovviamente no
- Il parametro è inizializzato all'atto della chiamata della funzione, e da quel momento non potrà più essere modificato
- Quindi:

```
int fun(const int j)
{
   j++ ;
}
```

ERRORE!! Non compila

Domanda

Il seguente programma è corretto?

```
void fun(int);

int main()
{
     fun(3) ;
     return 0;
}

void fun(const int j)
{
     cout<<j<<endl ;
}
```

Risposta

- No, perché il prototipo della funzione **`fun`** e l'intestazione della funzione **`fun`** nella definizione non coincidono
  - Nel prototipo manca il qualificatore **`const`**

Conclusione 1/2

- Vantaggi delle funzioni:
  - Testo del programma suddiviso in **unità significative**
  - Testo di ogni unità più breve
    - minore probabilità di errori
    - migliore verificabilità
  - Riutilizzo di codice
  - Migliore leggibilità
  - Supporto allo sviluppo **top-down** del software
    - Si può progettare prima quello che c'è da fare in generale, e poi si può realizzare ogni singola parte

Conclusione 2/2

- Come capiremo meglio in seguito, il vantaggio più grande è che le funzioni forniscono il **primo strumento per gestire la complessità**
  - Sono il meccanismo di base con cui, dato un problema più o meno complesso, lo si può spezzare in sotto-problemi distinti più semplici
  - Questa è di fatto **l'unica via** per risolvere problemi molto complessi